SIAM J. COMPUT.
Vol. 16, No. 1, February 1987

003

COMPLEXITY OF VIEWS: TREE AND CYCLIC SCHEMAS*

ODED SHMUELI† AND ALON ITAI†

Abstract. In relational databases a *view definition* is a query against the database, and a *view materialization* is the result of applying the view definition to the current database. A view materialization over a database may change as relations in the database undergo modifications.

Several problems concerning views are considered, many of which are shown to be hard (NP-complete or even $\sum_2^p$-complete). Each problem was treated for general databases and for the much simpler *tree* databases (also called *acyclic* databases).

View related problems over fixed schemas, in which only the data is allowed to vary, were examined. Methods to handle this case were presented; their complexity is polynomial: for tree schemas the degree of the polynomial is independent of the schema structure while for cyclic schemas the degree depends on the schema structure. These methods may present a practical possibility for dynamic view maintenance.

Key words. database, view, acyclic schema, trees, complexity, maintenance, amortized cost

AMS(MOS) subject classifications. 68P15, 68Q15, 68Q25

1. Introduction. A relational database [8], [27] is a collection of tables called *relations*, each containing a set of data rows called *tuples*. We differentiate between the structure of the database (the *schema*) and the time varying data (the *state*). A database schema $\mathbf{D}=(\mathbf{R}_1,\cdots,\mathbf{R}_n)$ is simply a multiset of finite subsets of a set (of *attributes*) $\mathbf{U}=\bigcup_{i=1}^{n}\mathbf{R}_i$; a schema can be viewed as the edge-set of a hypergraph over $\mathbf{U}$ [2].

One may partition the class of database schemas into *tree* and *cyclic* schemas. A schema is a tree schema if there is a tree whose nodes correspond to the schema's sets, and for each A in $\mathbf{U}$, the subtree induced by nodes containing A is connected. Tree schemas are also called *acyclic schemas.*

The partition above appears to be a good dividing line for database problem analyses. Acyclicity has wide implications in query processing [3], [6], [15], [17], [19], [27] and in dependency theory and schema design [4], [5], [11], [12], [20], [23]. Mathematical properties of acyclicity have also been studied [11], [16], [18], [19], [22], [26].

It has been shown [3], [6], [15], [28] that a class of queries (called *tree queries*), which imply acyclic databases, appears easier to process than queries which imply cyclic databases (called *cyclic queries*); and that the crux of query processing is constructing a tree (actually an "embedded tree") [17], [19]. The above results all hinge on the simple structure of tree schemas. In this paper we examine the relationship between schema structure and view related problems.

A *view definition* is a query against the database. A *view materialization* is the result of applying the view definition to the current database state. A view materialization over a database may change as relations in the database undergo modifications. When views are materialized, they remain valid as long as the underlying database remains unchanged. Usually, views are not materialized until needed. In certain systems views are never materialized; instead queries against the view are *modified* to reflect the view definition (a process called *query modification*).

The main difference between an "ordinary" query and a view definition has to do with the frequency of use. A view is either a query that is often posed or one which

* Received by the editors January 31, 1984; accepted for publication (in revised form) May 19, 1986.
† Computer Science Department, Technion-Israel Institute of Technology, Haifa, Israel.

delimits a relevant portion of the database for a group of users. Hence, maintaining a correct view materialization over time may prove beneficial.

The view definitions we consider are all of a simple form: perform the natural join of all the relations in the database and project the result on a set of attributes $\mathbf{X}$. This simple form actually encodes a much larger class of views [6]. We examine various problems associated with these views and their materialization maintenance over time. We note that view related problems were mainly treated in the past under the guise of query processing [9], [28].

View maintenance includes a variety of problems concerning the tuples in the view, equivalence of views, how changes in the underlying database affect the view and which kind of information is useful in maintaining a view. For example, one of the problems we treat is the following: Given a database D, a view definition $\mathbf{X}$, a tuple t and a relation schema $\mathbf{R}_i$ would the view materialization change when t is added to R_i?

Terminology is presented in § 2; our problem classification scheme is introduced, and a summary of the results is tabulated. Section 3 is devoted to "join problems," i.e., we consider the case $\mathbf{X}=\mathbf{U}$ (the view is the natural join of all the database relations). "Genuine" views, where $\mathbf{X}$ is a proper subset of $\mathbf{U}$, are treated in § 4. In § 5 we consider view complexity over a fixed schema. A preliminary version of the material in this section appeared in [25].

2. Terminology.

2.1. Relational databases. A *universe* $\mathbf{U}$ is a finite set of *attributes*. A *relation schema* $\mathbf{R}_i$ is a subset of $\mathbf{U}$, and a *database schema* $\mathbf{D}$ (or simply *schema*) is a multi-set of relation schemas.[1] Clearly, a database schema may be viewed as the set of edges of a hypergraph over $\mathbf{U}$ [2]. Associated with each $A\in\mathbf{U}$ is a possibly infinite *domain*, $\operatorname{dom}(A)$. The *domain* of a relation schema $\mathbf{R}_i=\{A_{i1},\cdots,A_{ih_i}\}$ is $\operatorname{dom}(\mathbf{R}_i)\overset{\text{def}}{=}\bigtimes_{k=1}^{h_i}\operatorname{dom}(A_{ik})$.

A *relation state* R_i for relation schema $\mathbf{R}_i$ is a finite subset of $\operatorname{dom}(\mathbf{R}_i)$; one can think about the state as a table of data with columns $A_1,\cdots,A_h$. A *database state* for schema $\mathbf{D}$ is an assignment of relation states to $\mathbf{D}$'s relation schemas. We use $\mathbf{D}=(\mathbf{R}_1,\cdots,\mathbf{R}_n)$ to denote a database schema and $D=(R_1,\cdots,R_n)$ for a corresponding state. Let $\bigcup(\mathbf{D})=\bigcup_{i=1}^{n}\mathbf{R}_i$.

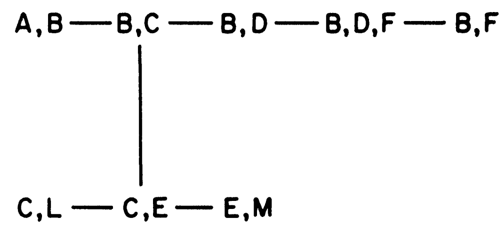
Elements in a relation state are called *tuples*. Tuple t over schema $\mathbf{R}$ *matches* tuple s over schema $\mathbf{S}$ if for all $A\in\mathbf{R}\cap\mathbf{S}$, the values of tuples t and s for attribute A are identical. The *projection* of relation state R over attribute set $\mathbf{X}\subseteq\mathbf{R}$, denoted $R[\mathbf{X}]$, is the maximal subset of $\operatorname{dom}(\mathbf{X})$ containing tuples that match some tuple in R. The (natural) *join* of relation states R and S, *denoted* $R\bowtie S$, is defined as the maximal subset in $\operatorname{dom}(\mathbf{R}\cup\mathbf{S})$ containing tuples that match a tuple in R and a tuple in S. A relation R is *total* in database D if it contains all its possible tuples composed of values appearing somewhere in the database, i.e. if $R=\bigtimes_{A\in\mathbf{R}}(\bigcup_{\mathbf{R}\in\mathbf{D}}R[A])$.

For a database D over schema $\mathbf{D}$ define $J(D)=\bowtie_{R\in D}R$; we use J instead of $J(D)$ when D is understood; define $J(R_i+t)$ to be the natural join of all the relations in D except that R_i is augmented with the tuple t. A *view definition* is simply a set $\mathbf{X}\subseteq\bigcup(\mathbf{D})$ of attributes; a *view materialization* V is $V=J[\mathbf{X}]$; also let $V(R_i+t)=J(R_i+t)[\mathbf{X}]$. Our class of views appears to be quite limited; however as is shown in [6] this class encodes a much larger class—those views defined by equijoin queries.

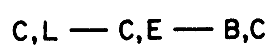
[1] All structures in this paper are finite.

2.2. Tree schemas. A *qual graph*[2] for **D** is an undirected graph whose nodes are in one-to-one correspondence with the relation schemas of **D**, such that for each attribute A, the subgraph induced by the nodes whose corresponding relation schemas contain A is connected [6]. **D** is a *tree schema* if some qual graph for it is a tree; otherwise **D** is a *cyclic schema.* See Fig. 2.1.

Consider the schema
$\mathbf{D}=(\{A, B\}, \{C, L\}, \{E, M\}, \{C, E\}, \{B, F\}, \{B, D, F\}, \{B, D\}, \{B, C\})$.
D is a tree schema viz.



For example, the subgraph induced by attribute C is:



The following is a cyclic schema:
$\mathbf{D}=(\{A, B\}, \{B, C\}, \{C, A\})$.
The only qual graph for **D** is

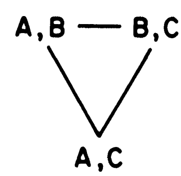


FIG. 2.1. *Tree and cyclic schemas.*

A database is a *tree database* (or an *acyclic database*) if the underlying database schema is acyclic; otherwise it is a *cyclic database.* The following simple procedure, discovered independently by [13] and [29], recognizes tree schemas. The procedure applies the following two steps until neither is applicable.

Step 1. Delete any attribute which appears in exactly one relation schema.

Step 2. Find two relation schemas **R** and **S** in **D** such that $\mathbf{R} \subseteq \mathbf{S}$; delete **R** from **D**.
It can be shown that the original schema is a tree schema iff upon termination of the above procedure the database schema consists of a single (empty) relation schema. (A linear time algorithm for recognizing tree schemas appears in [26].)

2.3. Complexity classes. A problem, or a language, L is in $(NP)P$ if given a string x, determining $x \in L$ can be done by a (non)deterministic Turing machine within time polynomial in the size of the input x. A problem is in Σ_2^p if it can be solved by a nondeterministic Turing machine, which may use an oracle for a set in NP, in time polynomial in the size of the input. (An oracle for a language L can be thought of as a "subroutine" which when given some string x answers in one time unit "yes" if $x \in L$ and "no" otherwise. The subtle point is that the Turing machine can make use of the fact that a string does not belong to the oracle set.) For more details, the reader is referred to [14].

[2] We use traditional graph theory notation.

A problem A is *polynomial-time complete* for the complexity class C, or *C-complete* for short, if for any other problem B in C, there is a polynomial time bounded Turing machine M which transforms a string x into a string $M(x)$, such that $x \in B$ iff $M(x) \in A$. Intuitively, if A is a complete problem in a class, then a polynomial algorithm for solving A will provide an efficient algorithm for *all* problems in the class.

Several known polynomial-time complete problems are:

(1) 3SAT. Given: a boolean formula in 3CNF, i.e. in conjunctive normal form having three literals per clause [1].
Question: Is F satisfiable? That is, is there an assignment to the boolean variables in F which makes it *true*?
3SAT is NP-complete.

(2) Let L be the language

$$L=\{F(X, Y) \mid \exists X\ \forall Y\, F(X, Y) \text{ is } \mathit{true}\}.$$

L is Σ_2^p-complete (see [14]).

2.4. Problem classification. In the following definition the size of a database is the size of the schema plus the size of the state. We shall classify problems according to the following criteria:

(1) The object in question:
J —A problem concerning the join of a given database D.
V —A problem concerning the view of a given database D on the given attributes $\mathbf{X}$.

(2) The type of data supplied (optional).
C —Change: given a tuple t and an index i, consider the new join $J(R_i+t)$ (or new view $V(R_i+t)$).
G —Given: the input consists of the input to C *and* the old join (or old view materialization).

(3) The question:
E —Emptiness: is the join (or the view) *not* empty?
M—Membership: given a tuple t does t belong to the join (view)?
I —Intotality: is the join (view) *not* total?
N —Not equal: is the new join (view) not equal to the old one? (This question is meaningful only if C or G are present.)

For example: the problem JE is defined as follows:

Given a database D is $J \neq \varnothing$?

And the problem VGN is:

Given a database D, a view definition $\mathbf{X}$, the view materialization V, an index i and the tuple t is $V \neq V(R_i+t)$?

Thus a total of 22 different problems are defined, results concerning these problems are shown in Table 2.1. (NA appears when the problem is not defined.) If $\mathbf{D}$ is a tree schema the above problems are sometimes easier as seen in Table 2.2.

3. Join problems.

3.1. Polynomial problems.

JM, JCM *and* JGM. Checking $t \in J$ amounts to checking that for all database relations R_i, $i=1, \cdots, n$, $t[\mathbf{R}_i] \in R_i$.

TABLE 2.1

	E	M	I	N
J	NP-C[14], [21]	P	P	NA
JC	NP-C	P	P	NP-C
JG	NP-C	P	P	NP-C
V	NP-C[14], [21]	NP-C[14], [28]	Σ_2^p-C	NA
VC	NP-C	NP-C	Σ_2^p-C	Σ_2^p-C
VG	NP-C	NP-C	Σ_2^p-C	NP-C

TABLE 2.2

	E	M	I	N
J	P*	P	P	NA
JC	P*	P	P	P*
JG	P*	P	P	P*
V	P	P[28]	NP-C	NA
VC	P	P	NP-C	NP-C
VG	P	P	NP-C	P

* The results of [3], [6] imply that these problems are polynomial.

Remark. For tree databases all the problems are polynomial if the view definition $\mathbf{X}$ is contained in any one of the relations $\mathbf{R}_i$.

JI, JCI *and* JGI. If the join is total then each R_i must be total. Conversely, if for all i R_i is total then the join is total. Therefore, the join is total iff each R_i is total. The latter condition can be checked by counting the number of tuples in each R_i.

3.2. NP-complete problems.

JE. This problem was first shown to be NP-complete by Chandra and Merlin [7]. The problem is in NP since a nondeterministic Turing machine can guess a tuple s and check whether $s \in J$ in polynomial time. We show completeness by using the following construction.

Standard database construction. Given a boolean formula $F(X)$ in 3CNF, we show how to construct a database D such that $J \neq \emptyset$ iff F is satisfiable. With each clause of F we associate a relation schema, whose attributes are the variables appearing in the clause. W.l.o.g. we may assume that each relation schema thus constructed consists of three attributes. Each relation consists of the seven boolean assignments which make the original clause evaluate to *true*. It can be easily seen that each tuple in the natural join of all the relations in the database "spells out" an assignment to F's variables satisfying F. Hence, $J \neq \emptyset$ iff F is satisfiable. Observe that the size of the database constructed above is linear in the size of the formula.

JCE. The problem is in NP since it suffices to guess a tuple s and check whether it belongs to $J(R_i + t)$. The completeness follows by a reduction from JE: Let $D = (R_1, \cdots, R_k)$ be an instance of JE. Let $\mathbf{D}' = (\mathbf{R}_0', \mathbf{R}_1', \cdots, \mathbf{R}_k')$ where $\mathbf{R}_0'$ consists of a single attribute C, $R_0' = \{\langle b \rangle\}$ and the attributes of $\mathbf{R}_i'$ are those of $\mathbf{R}_i$ and the additional attribute C. Construct D$'$ with

$$R_i' = \{\langle a_1, \cdots, a_{h_i}, a\rangle | \langle a_1, \cdots, a_{hi}\rangle \in R_i\}.$$

By construction $J(D')=\varnothing$. We augment R_0' by the tuple $t=\langle a\rangle$, clearly

$$J(R_0'+t)\neq\varnothing \text{ iff } J(D)\neq\varnothing.$$

JGE. In the previous construction the old join was empty; thus we may assume it was given.

JCN *and* JGN. The problem is in NP since a nondeterministic Turing machine can guess a tuple s such that $s\notin J$ and $s\in J(R_i+t)$. (Recall that JM is polynomial.) Completeness follows by a reduction from JE (as in the completeness proof of JCE—we add a new column C and a new relation $\mathbf{R}_0$ such that $J(D')=\varnothing$: however after adding a new tuple t $J(R_0+t)=\varnothing$ iff $J(D)=\varnothing$).

4. View problems.

4.1. The VI problem over general databases. Let $F(X)$ be a boolean formula, and t a global truth assignment, i.e., for each variable y_k, $t(y_k)=$ *true* or $t(y_k)=$ *false*. Define $t(F)=F(t(X))$ to be the value of F under the assignment t, where $t((x_1,\cdots,x_n))=(t(x_1),\cdots,t(x_n))$.

The structure of F defines a tree T_F, the root of which is the main connective of F and whose children are the subtrees associated with the arguments of the connective. Each leaf of the tree is associated with a variable $x_j\in X$. Let T_F have m nodes, numbered $1,\cdots,m$. A distinct variable z_i is associated with each node. Let $Z=\{z_1.\cdots,z_m\}$. Let F_i be the subformula associated with the node i.

We define a formula G_i associated with each node: For a leaf x_j, $G_i=z_i\equiv x_j$, expressed in CNF (conjunctive normal form)

$$G_i=(x_j+\bar z_i)\cdot(\bar x_j+z_i).$$

For a "not" node whose child is z_j, $G_i=z_i\equiv\bar x_j$, or in CNF

$$G_i=(x_j+z_i)\cdot(\bar x_j+\bar z_i).$$

For an "and" node whose children are z_L and z_R, $G_i=z_i\equiv z_L\cdot z_R$, or in CNF

$$G_i=(z_i+\bar z_L+\bar z_R)\cdot(\bar z_i+z_L)\cdot(\bar z_i+z_R).$$

Finally, for an "or" node whose children are z_L and z_R, $G_i=z_i\equiv z_L+z_R$, or in CNF

$$G_i=(\bar z_i+z_L+z_R)\cdot(z_i+\bar z_L)\cdot(z_i+\bar z_R).$$

Let $G_F(X,Z)=\Lambda_{i\in T_F}G_i$.

LEMMA 4.1. *Let t be a truth assignment; then $t(G_F(X,Z))$ is true iff for all i $t(z_i)=t(F_i(X))$.*

Proof. By induction on the number of nodes of T_F.

Basis. If T_F has size 1, then $F=x_j$. Therefore, $Z=\{z_i\}$ and $G_F=z_i\equiv x_j$. Because of the structure of G_F:

$$G_F(t(X,Z))=\textit{true} \quad \text{iff } t(z_i)=t(x_j).$$

Since $t(x_j)=t(F)$:

$$G_F(t(X),Z)=\textit{true} \quad \text{iff } t(z_i)=t(F).$$

Induction step. Suppose, for instance, that $F(X)=F_L+F_R$. Therefore, $G_F=(\Lambda_{i\in F_{T_L}}G_i\Lambda_{i\in F_{T_R}}G_i)\cdot(z_F\equiv(z_L+z_R))$. Since $G_L=\Lambda_{i\in F_{T_L}}G_i$ and $G_R=\Lambda_{i\in F_{T_R}}G_i$, $G_F=G_L\cdot G_R\cdot(z_F\equiv(z_L+z_R))$.

For $t(G_F)$ to be *true*, $t(G_L)=$ *true*, $t(G_R)=$ *true* and $t(z_F\equiv(z_L+z_R))=$ *true*. By induction, $t(z_k)=t(F_k)$ for all variables z_k in either the left or the right subtree. Obviously, $t(z_F)=$ *true* iff at least one of z_L or z_R has the value *true*. W.l.o.g., z_L is *true* and by the induction hypothesis, $t(F_L) = true$, which implies that $t(F) = true$. Hence, $t(z_F) = t(F)$. The case $t(z_F) = false$ is handled similarly.

Conversely, if $t(G_F)$ is *false* then $t(G_L) = false$ or $t(G_R) = false$ or $t(z_F \equiv z_L + z_R) = false$. If one of the first two cases holds, then by the induction hypothesis there exists a z_i such that $t(z_i) \neq t(F_i)$. Otherwise, $t(z_F) \neq t(z_L + z_R)$ and since $t(G_L) = t(G_R) = true$, $t(z_L) = t(F_L) = true$ and $t(z_R) = t(F_R) = true$. Consequently, $false = t(z_F) \neq t(F) = true$.

The other cases are similar. □

The above lemma can also be proved by using the techniques of Cook's theorem [1].

LEMMA 4.2. *For every Boolean formula $F(X)$ over the variables $X = \{x_1, \cdots, x_n\}$ there exists a* 3CNF *formula $H_F(X, Z)$ $(Z = \{z_1, \cdots, z_m\})$ such that*

(i) *For all assignments t to the variables X*

$$F(t(X)) = true \quad \textit{iff } H_F(t(X, Z)) \textit{ is satisfiable};$$

(ii) *The size of H_F is linearly bounded by the size of F.*

Proof. Construct G_F as in Lemma 4.1. Using the above notation, let $H_F(X, Z) = G_F(X, Z) \cdot z_F$.

Let t satisfy H_F, i.e., $H_F(t(X), t(Z)) = true$. Therefore, both $t(z_F) = true$ and $t(G_F(X, Z)) = G_F(t(X), t(Z)) = true$. By Lemma 4.1, if $G_F(t(X), t(Z)) = true$, $t(z_F) = F(t(X))$. Since $t(z_F) = true$, $F(t(X)) = true$.

Conversely, suppose $F(t(X)) = true$. Let t assign the value $F_i(t(X))$ to z_i. By Lemma 4.1 $G_F(t(X), t(Z)) = true$, and since by the assignment $t(z_F) = F(t(X)) = true$, $H_F(t(X), t(Z)) = true$ and hence $H_F(t(X), Z)$ is satisfiable. □

THEOREM 4.1. VI *is Σ_2^p-complete.*

Proof. The fact that VI is in Σ_2^p is straightforward: a nondeterministic Turing machine M can guess v and then consult an oracle for $v \notin V$, the oracle set is in NP since determining whether $v \in V$ is NP-complete.

Let L be the language

$$L = \{F(X, Y) | \exists X \forall Y F(X, Y) \text{ is } \textit{true}\}.$$

L is complete in Σ_2^p (see [14]). Given a string of the form $F(X, Y)$ we show how to construct a database D and view definition X, such that $\exists X \forall Y F(X, Y)$ is *true* iff $(\exists v v \notin V)$.

By Lemma 4.2, given a boolean formula $F(X, Y)$ we construct, in linear time in the size of F, a boolean formula $H_{\neg F}(X, Y, Z)$ in 3CNF such that for all boolean vectors t_X, t_Y,

$$[\exists Z H_{\neg F}(t_X, t_Y, Z)] \leftrightarrow [\neg F(t_X, t_Y)] \text{ is } \textit{true},$$

which implies that for any boolean vector t_Z,

$$[H_{\neg F}(t_X, t_Y, t_Z)] \rightarrow \neg F(t_X, t_Y) \text{ is } \textit{true}.$$

Build a standard database D over $\mathbf{U} = X \cup Y \cup Z$ for $H_{\neg F}$ as described in § 3. Let X be the view definition on D.

CLAIM. $[\exists v v \notin V]$ *iff the formula* $[\exists X \forall Y F(X, Y)]$ *is true.*

Proof of claim. Assume $[\exists v v \notin V]$: Each tuple t over $\mathbf{U}$ "spells out" an assignment t_X, t_Y, t_Z to the boolean variables in $H_{\neg F}(X, Y, Z)$. Suppose $v \notin V$, then for all tuples t such that $t_X = v$ $H_{\neg F}(t_X, t_Y, t_Z) = false$ (otherwise, if $H_{\neg F}(t_X, t_Y, t_Z) = true$ then t would be in J and v in V). Let v spell out the assignment t_X to the X variables in $H_{\neg F}(X, Y, Z)$. It follows that

$$\forall Y \forall Z \quad [H_{\neg F}(t_X, Y, Z) = false].$$

In other words

$$\forall Y \forall Z \quad \neg H_{\neg F}(t_X, Y, Z).$$

This gives

$$\forall Y \quad \neg[\exists Z H_{\neg F}(t_X, Y, Z)].$$

Consider any boolean vector t_Y. By the above we have

$$\neg[\exists Z H_{\neg F}(t_X, t_Y, Z)].$$

But, by Lemma 4.1

$$[(\exists Z) H_{\neg F}(t_X, t_Y, Z)] \leftrightarrow [\neg F(t_X, t_Y)].$$

We conclude that for any boolean vector t_Y

$$\neg\neg F(t_X, t_Y) \quad \text{or equivalently } F(t_Y, t_X),$$

which means that indeed $\exists X$ (namely t_X) such that for all Y, $F(X, Y)$.

We now assume $[\exists X \forall Y F(X, Y)]$:

Let t_X be a boolean vector such that $\forall Y F(t_X, Y)$. Choosing any t_Y, we have $F(t_X, t_Y)$. By Lemma 4.2,

$$\neg\exists Z \quad H_{\neg F}(t_X, t_Y, Z)$$

i.e.,

$$\forall Z \quad \neg H_{\neg F}(t_X, t_Y, Z).$$

Since t_Y was chosen arbitrarily it follows that

$$[\forall Y \forall Z \neg H_{\neg F}(t_X, Y, Z)] = \textit{true}. \tag{4.1}$$

We now show that (4.1) implies $\exists v v \notin V$. It suffices to show that

$$[\neg\exists v v \notin V] \rightarrow \neg[\forall Y \forall Z \neg H_{\neg F}(t_X, Y, Z)],$$

or, equivalently, that $[\forall v v \in V] \rightarrow \neg[\forall Y \forall Z \neg H_{\neg F}(t_X, Y, Z)]$. In other words, we have to show that

$$[\forall v v \in V] \rightarrow \exists Y \exists Z H_{\neg F}(t_X, Y, Z).$$

From $\forall v v \in V$ follows the existence of a boolean vector t_X in V. But if t_X is a tuple in the view, then there must exist a "parent" tuple $t = (t_X, t_Y, t_Z)$ such that $t_X = t[X]$. The existence of t implies the existence of t_Y and t_Z such that $H_{\neg F}(t_X, t_Y, t_Z) = \textit{true}$. But this simply states that

$$\exists Y \exists Z \quad H_{\neg F}(t_X, Y, Z) \quad \square$$

4.2. The VI problem over tree databases. The following problems are useful in proving the NP-completeness result:

SET COVER:

Given n sets $C_1, \cdots, C_n$ and an integer k, are there $C_{i_1}, \cdots, C_{i_k}$ whose union equals $\bigcup_{i=1}^{n} C_i$? (NP-complete [14])?

FAMILY COVER:

Given $k > 1$ families $S_1, \cdots, S_k$, where each S_i contains n sets $S_{i1}, \cdots, S_{in}$, are there $S_{1i_1}, \cdots, S_{ki_k}$ such that $\bigcup_{j=1}^{k} S_{ji_j} = \bigcup_{i=1}^{k} \bigcup_{j=1}^{n} S_{ij}$?

NCP (Non-Cartesian Product):

Given families $F_1, \cdots, F_h$, where $F_i = (C_{i1}, \cdots, C_{ik})$, $(k > 1)$, does the NCP inequality condition

$$\bigcup_{i=1}^{h}(C_{i1} \times \cdots \times C_{ik}) \subsetneq \left(\bigcup_{i=1}^{h} C_{i1}\right) \times \cdots \times \left(\bigcup_{i=1}^{h} C_{ik}\right)$$

hold?

CLAIM 1. FAMILY COVER *is* NP-*complete.*

Proof. Let $C_1, \cdots, C_n$ and $k > 1$ be an instance of SET COVER. Form an instance of FAMILY COVER with families $S_1, \cdots, S_k$, where each S_i contains a copy of $C_1, \cdots, C_n$.

CLAIM 2. NCP *is* NP-*complete.*

Proof. The problem is obviously in NP. Let $S_1, \cdots, S_k$ be an instance of FAMILY COVER, where $S_i = \{S_{i1}, \cdots, S_{in}\}$. W.l.o.g. each S_i has a unique element associated with it that is a member of each set in the S_i family, and is *not* a member of any set in any other family. Let $A = \bigcup_{i=1}^{k} \bigcup_{j=1}^{n} S_{ij}$. We form an instance of NCP by associating with each $a \in A$ a family of sets $F_a = \{F_{a1}, \cdots, F_{ak}\}$ where $F_{aj} = \{\langle j, p\rangle | a \notin S_{jp}$ and $1 \leqq p \leqq n\}$.

We claim that there is a FAMILY COVER iff the NCP inequality condition is satisfied.

First, observe that $(\bigcup_{a\in A} F_{a1}) \times \cdots \times (\bigcup_{a \in A} F_{ak}) = \bigtimes_{i=1}^{k}\{\langle i, 1\rangle, \cdots, \langle i, n\rangle\}$, because there is a unique element associated with each family S_i which appears in no other family S_j.

Suppose there is a family cover $S_{1i_1}, \cdots, S_{ki_k}$. We claim that

$$t = \langle\langle 1, i_1\rangle, \cdots, \langle k, i_k\rangle\rangle \notin \bigcup_{a \in A}(F_{a1} \times \cdots \times F_{ak}).$$

Otherwise, there must exist $a \in A$ such that $t \in (F_{a1} \times \cdots \times F_{ak})$; i.e. $\langle 1, i_1\rangle \in F_{a1}, \cdots, \langle k, i_k\rangle \in F_{ak}$, which means $a \notin S_{1i_1}, \cdots, a \notin S_{ki_k}$, which in turn implies that $S_{1i_1}, \cdots, S_{ki_k}$ is *not* a FAMILY COVER. This is a contradiction.

Suppose that $t = \langle\langle 1, i_1\rangle, \cdots, \langle k, i_k\rangle\rangle \notin \bigcup_{a \in A}(F_{a1} \times \cdots \times F_{ak})$. We claim that $S_{1i_1}, \cdots, S_{ki_k}$ is a FAMILY COVER. Consider $a \in A$. Since $t \notin (F_{a1} \times \cdots \times F_{ak})$, it follows that for some $1 \leqq j \leqq k$, $\langle j, i_j\rangle \notin F_{aj}$, i.e. $a \in S_{ji_j}$. This reasoning holds for all $a \in A$ and hence $S_{1i_1}, \cdots, S_{ki_k}$ is a FAMILY COVER. □

THEOREM 4.2. *For tree schemas,* VI *is* NP-*complete.*

Proof. For tree schemas $t \in V$ is in P, hence the problem is in NP. Consider an NCP instance $F_1, \cdots, F_h$ with $F_i = (F_{i1}, \cdots, F_{ik})$. We now show how to build a database and view definition (forming a VI instance) such that the NCP inequality holds for the NCP instance iff there is a tuple which is not in the materialized view of the VI instance.

Construct $k+1$ relations $\mathbf{R}_0, \mathbf{R}_1, \cdots, \mathbf{R}_k$. C is the sole attribute of $\mathbf{R}_0$. For $1 \leqq i \leqq k$, $\mathbf{R}_i$ has two attributes: C and B_i. These relations constitute a tree database, whose root node corresponds to $\mathbf{R}_0$. The database state is constructed as follows:

$$R_0 = \{\langle i\rangle | i = 1, \cdots, k\},$$

$$R_m = \{\langle i, e\rangle | e \in F_{im}\}, \qquad m = 1, \cdots, k, \quad i = 1, \cdots, h.$$

Clearly,

$$V = (R_0 \bowtie \cdots \bowtie R_k)[B_1 \cdots B_k] = \bigcup_{i=1}^{h}(F_{i1} \times \cdots \times F_{ik}).$$

So, if there is $v \notin V$ then the NCP inequality holds. Conversely, if the NCP inequality holds, then there is a tuple $v \notin V$. □

4.3. The VCI and VGI problems. Given a database D, a view definition $\mathbf{X}$ and a tuple t, the VCI problem is to determine whether there exists $v \notin V(R_i + t)$. The problem is easily seen in Σ_2^p for general databases and in NP for tree databases; we show that it is complete in these respective cases.

The proof is done by reducing a VI instance to a VCI instance. The reduction is by adding a new column, say N, to $\mathbf{R}_1$ and setting each tuple entry in this column to a. Also, add a new relation $\mathbf{R}_N$, whose sole column is N, containing no tuples. Clearly, J on this new database is empty and so is V. However, adding $\langle a \rangle$ to R_N will yield the original view. Thus, there exists $v \notin V$ in the original database iff there exists $v \notin V(R_N + \langle a \rangle)$.

The reduction shows that VCI is Σ_2^p-complete for general databases. Observe that if the original database is a tree database, then so is the new database. Hence, VCI is NP-complete for tree databases.

Since in the previous reductions the view materialization was empty before adding $\langle a \rangle$, the same results hold for VGI.

4.4. The VCN problem. The VCN problem is to determine whether $V(R_i + t) \neq V$. This problem is clearly in Σ_2^p since a nondeterministic Turing machine can guess a "new" join tuple, extract from it a "supposed" new view tuple, and consult an oracle as to whether the new view tuple belongs to the original view. We show that VCN is Σ_2^p-complete by reducing VI, which was previously shown Σ_2^p-complete, to VCN.

The VI completeness proof actually yields that VI over databases with at most three attributes per relation schema, whose attributes all have the domain $\{true, false\}$, is Σ_2^p-complete. Given such a database D and some view definition $\mathbf{X}$, we reduce VI to VCN as follows. We construct a database $\mathbf{D}'$ by adding to each relation schema in $\mathbf{D}$ a new column, say N. We also add a new relation $\mathbf{R}_N$ with the sole attribute N.

The database D' is populated thus: To each of the original tuples in D the N column entry is set to a. We also put the tuple $\langle a \rangle$ in R_N. To each relation in D' we add all the eight $\{true, false\}$ combinations for the original columns, with the N column entry set to b.

Note that, by construction, V over D is identical to V over D'; the a values preserve the original view and the b values add nothing as b does not appear in R_N. We claim that $V \neq V(R_N + \langle b \rangle)$ in D' iff $\exists v(v \notin V)$ in D. Since all $\{true, false\}$ combinations are present in D' in conjunction with the N column value b, the addition of $\langle b \rangle$ to R_N will make the view total. So, if the view with $\langle b \rangle$ added is different than the view without $\langle b \rangle$, we conclude that the original view "missed" a tuple. Conversely, if the view prior to $\langle b \rangle$'s addition "missed" a tuple, certainly $V \neq V(R_N + \langle b \rangle)$ following $\langle b \rangle$'s addition. Hence we have proved the following theorem.

THEOREM 4.3. VCN *is* Σ_2^p*-complete.*

We now treat the VCN problem over tree databases.

THEOREM 4.4. *For tree schemas, the* VCN *problem is* NP*-complete.*

Proof. For tree schemas $t \in V$ is in P; hence VCN is in NP. By Theorem 4.2, VI is NP-complete over tree databases. In addition the reduction from VI to VCN presented above preserves the tree property of the schema because attribute N is uniformly added to each relation. Hence VCN is NP-complete over tree databases. □

4.5. The VCE, VGE and VGN problems. In § 3 it was proved that JGE, JCN and JGN are NP-complete over general databases. The corresponding view problems: VGE, VCN and VGN, are therefore also NP-complete (with $\mathbf{X} \equiv \mathbf{U}$). For tree databases VE is polynomial, this is because V is empty iff J is empty, and JE is polynomial for tree databases. It follows that VCE and VGE are also polynomial.

THEOREM 4.5. *For tree schemas* VGN *is in* P.

Proof. Yannakakis has shown that the view materialization of a tree database can be computed by an algorithm which is polynomial in the size of the database and the size of the view materialization [28]: Let p be that polynomial.

Let D' be the database resulting from the addition of t to R_i and $V' = V(R_i + t)$. We now start producing V' from D' using Yannakakis' algorithm and abort the algorithm if it does not halt within $p(|D|+|t|, |V|)$ steps. Since $V' \supseteq V$, the views are different iff $|V'| > |V|$. If $V' = V$ then Yannakakis' algorithm must finish within $p(|D'|, |V'|) = p(|D|+|t|, |V|)$ steps. Thus if the algorithm aborted, then the views are different. On the other hand, if the algorithm terminated, then we have produced the materialization V' and can easily check whether $|V| = |V'|$. The entire procedure requires essentially $p(|D|+|t|, |V|)$ steps. □

5. Fixed schemas.[3] In the problems previously analyzed, the database schema was part of the problem instance. This section treats the case in which the database schema is *fixed* over all problem instances, i.e., problem instances differ only in the tuples in the relations.

5.1. Additions into a tree database with the view contained in one of the relations. Let us consider a special case. Suppose for some $r \leqq n$, $\mathbf{X} \subseteq \mathbf{R}_r$. Furthermore, assume that relations $\mathbf{R}_1, \cdots, \mathbf{R}_k$ constitute a tree schema. Let T be a qual tree with $\mathbf{R}_r$ at its root. ($\mathbf{R}_r$ is called the *root* relation and the relations at the leaves are called *leaf* relations.)

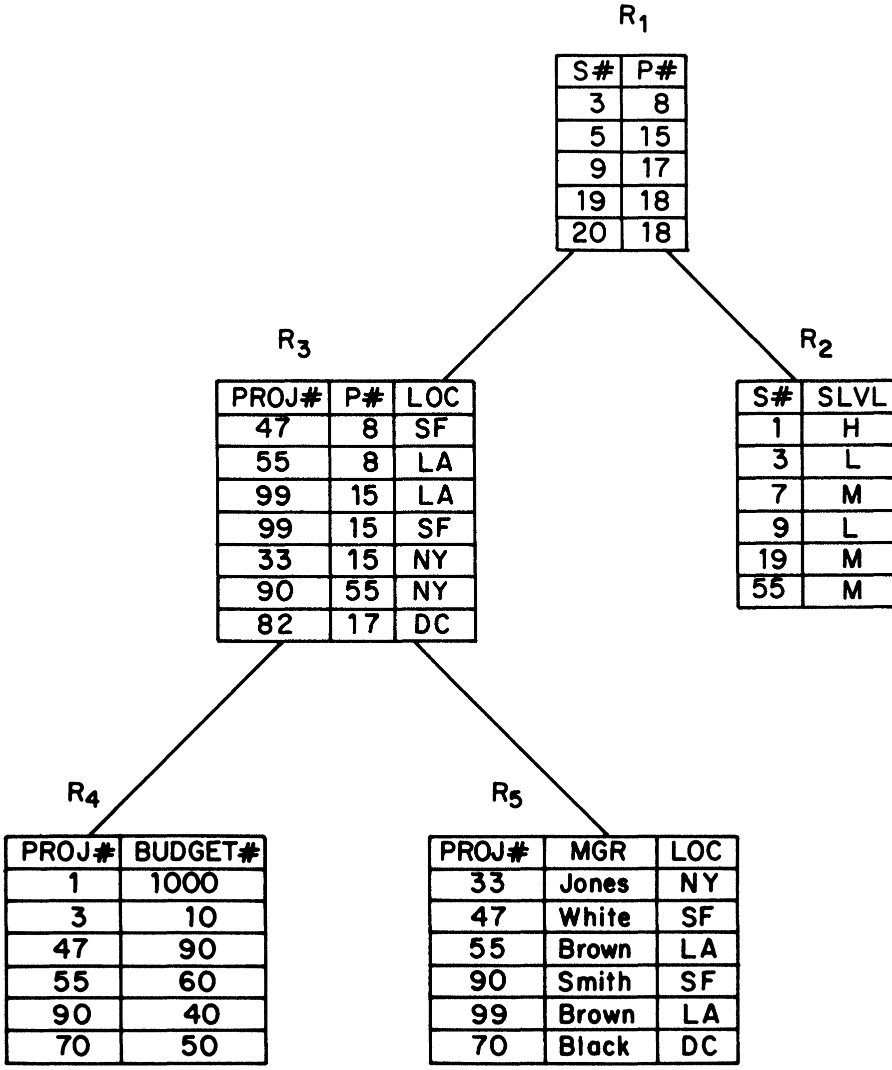
Let $\mathbf{R}_i$ be a node in T and $\mathbf{R}_j$ its child. Tuple $t \in R_i$ is *supported* by tuple $s \in R_j$ if t matches s. A tuple $t \in R_i$ is *good* if every child R_j of R_i has a good tuple $s \in R_j$ which supports t. Also, all tuples in a leaf relation are considered good. Tuple $t \in R_i$ is *compatible below* with a child relation R_j if there is a good tuple $s_j \in R_j$ which supports t. Hence, $t \in R_i$ is good iff t is compatible below with all of $\mathbf{R}_i$'s children.

Intuitively, a tuple $t \in R_i$ is good if it is unanimously supported by all its children, its children's children and so on, i.e. t belongs to the projection onto $\mathbf{R}_i$ of all the relations in the subtree rooted at $\mathbf{R}_i$. Observe that $t \in R_i$ may contribute to $J(D)$, and therefore possibly to V, iff t is good. In other words, all nongood tuples, which we call *bad*, will definitely not contribute to $J(D)$ and V.

Consider the database of Fig. 5.1, with $\mathbf{X} \subseteq \mathbf{R}_1$. The relations $\mathbf{R}_2$, $\mathbf{R}_4$ and $\mathbf{R}_5$ are leaf relations and therefore all their tuples are good. Only the first two tuples of R_3 are good (e.g., $\langle 47, 8, \text{SF}\rangle$ matches $\langle 47, 90\rangle \in R_4$ and $\langle 47, \text{White}, \text{SF}\rangle \in R_5$; and $\langle 99, 15, \text{LA}\rangle$ matches $\langle 99, \text{Brown}, \text{LA}\rangle \in R_5$ but no tuple of R_4). Tuple $\langle 3, 8\rangle$ is the only good tuple of R_1, it matches the good tuples $\langle 47, 8, \text{SF}\rangle \in R_3$ and $\langle 3, \text{L}\rangle \in R_2$. Tuple $\langle 9, 17\rangle \in R_1$ is bad because *all* the R_3 tuples it matches are bad.

The partition of each original relation into a good part and a bad part is helpful when processing updates. We start by discussing tuple addition into the tree database of Fig. 5.1. There are three cases to consider—the relation is a root, a leaf or an internal node.

(i) *Root.* Suppose $t_1 = \langle 9, 8\rangle$ is added to R_1 to indicate that supplier number 9 now supplies part 8. Tuple t_1 is good since it is supported by the good tuples $\langle 55, 8, \text{LA}\rangle \in R_3$ which indicates that project 55, located at LA, requires part number 5, and $\langle 9, \text{L}\rangle \in R_2$ indicating that the service level of supplier number 9 is rated L. On the other hand,



R_1: supplier $S\#$ supplies part P#;
R_2: each supplier may provide product support (indicated by SLEVEL);
R_3: project PROJ# may need part P# at location LOC;
R_4: project PROJ# has an allocated BUDGET;
R_5: project PROJ# is managed by MGR at location LOC.
The view is on $S\#$ and $P\#$.

FIG. 5.1. *An example tree database.*

adding the tuple $t_2 = \langle 7, 99 \rangle$ to R_1 cannot possibly change the view since it is not supported by any good tuple of R_3. (The fact that it is supported by the good tuple $\langle 7, \mathrm{M} \rangle \in R_2$ is immaterial.) Thus t_2 should be added to bad(R_1).

(ii) *Leaf.* Suppose $t_3 = \langle 99, 30 \rangle$ is added to R_4 indicating that project 99 has been assigned a budget 30K. First, leaves only have good parts. Thus t_3 is added to good(R_4). Now, it is possible that the new addition may change the good part of R_3 (which is equivalent to changing an internal node and is discussed below). Namely, $t_4 = \langle 99, 15, \mathrm{LA} \rangle \in R_3$, previously supported only by the good tuple $\langle 99, \mathrm{Brown}, \mathrm{LA} \rangle \in R_5$ is now also supported by t_3; thus t_4 should move to the good part of R_3. This effect might propagate up the tree. On the other hand ⟨99, 15, SF⟩, which also matches t_3, remains in bad(R_3) since even now it is not supported by any R_5-tuple. To summarize, if the new tuple is good, we should check the matching tuples in the bad part of the parent node because now some of them can become good.

(iii) *Internal node.* Suppose $t_5 = \langle 70, 18, \text{DC}\rangle$ is added to R_3. Tuple t_5 is good since it is supported both by $\langle 70, 50\rangle \in R_4$ and by $\langle 70, \text{Black}, \text{DC}\rangle \in R_5$. As mentioned above, changes to an internal node may propagate upwards. We now have to check if t_5 is *compatible above*—i.e. matches with tuples in its parent relation. Indeed, t_5 matches $t_6 = \langle 19, 18\rangle$ and $t_7 = \langle 20, 18\rangle$ of R_1. Hence t_6 becomes good since it is supported by $\langle 19, \text{M}\rangle \in R_2$, while t_7 remains bad since it is not supported by any (good) R_2-tuple.

Consider an empty database over our fixed schema. To this state apply a sequence of n tuple additions (into various relations). Throughout this addition process maintain the database as above—i.e. with good-bad partitions. Compatibility above is checked only when a tuple becomes good. A tuple t is thus compared to all tuples in its parent node, and if we find a matching bad tuple s then s is checked for compatibility since potentially s may have become good. Thus, each time a tuple becomes good it initiates $O(n)$ compatibility checks. Each compatibility check compares a tuple with all the tuples in a parent (or child) node. Thus, in the worst case, each tuple is compared to all other tuples, costing $O(n^2)$ time. Thus, the cost of n additions in this naive scheme is $O(n^3)$.

The following *good-bad marking scheme* reduces the number of times t is checked for compatability below. Consider a tuple t in bad(R_3) (see Fig. 5.1). It may be there because either

(i) $t[\text{PROJ}\#]$ is not mentioned in R_4, or

(ii) $t[\text{PROJ}\#, \text{LOC}]$ is not mentioned in R_5.

However, we have no information as to which of these cases hold. To remedy this situation, with each tuple in bad(R_3) we associate *marks*. For example, an R_4-mark would indicate that t could find no match in R_4; likewise, for an R_5-mark. As relations change marks may need updating.

Data structures. We now describe the data structures employed and how the insert and delete operations are performed. Consider relation (node) R_i with tree parent R_p and children $R_1, \cdots, R_c$. Define $\mathbf{Z}_{im} = \mathbf{R}_i \cap \mathbf{R}_m$. The following balanced trees[4] are associated with R_i:

(a) For each child R_m, a tree C_{im} containing all tuples of $R_i[\mathbf{Z}_{im}]$. For each $w_m \in C_{im}$ we associate the list of tuples t in R_i with $w_m = t[\mathbf{Z}_{im}]$ and a *good-counter* indicating the number of good tuples (in R_m) that support it.

(b) T_i-containing all the tuples (good and bad) of R_i; each tuple has a *mark-counter*, which counts the number of bad marks it has, and a pointer (called the *up-pointer*) to the tuple $v = t[\mathbf{Z}_{pi}] \in C_{pi}$. ($t$ has an R_m-mark iff w_m's good-counter is equal to zero.)

We should note that in all the appearances of t in these trees, it is the *same t*, i.e., t has a record structure which allows it to concurrently be a part of several lists.

Operations. Consider a tuple t in R_i with $v = t[\mathbf{Z}_{pi}]$ and $w_m = t[\mathbf{Z}_{im}]$.

Insert (t, R_i)

A. First, t is inserted into the tree T_i and its mark-counter is set to zero.

B. For each child R_m treat C_m as follows:

[4] On a set with n elements, the operations insert, delete and member can be performed in $O(\log n)$ when the set is implemented as a balanced tree; examples for balanced tree schemes include AVL trees and 2:3 trees [1].

(a) If w_m does not appear in C_{im} then insert it into C_{im} and set its good-counter to zero. Add t to w_m's list and if w_m's good-counter $=0$ (i.e. w_m is bad) then add 1 to t's mark-counter, thus counting the number of children of R_i that do not support t.

(b) Once all C_{im}'s have been treated, if t's mark-counter $=0$ then t is bad and we are done; otherwise t is good and we set t's up-pointer to v's appearance in C_{pi}. (Of course, if v does not appear in C_{pi} then it is inserted.) Finally, v's good-counter in C_{pi} is incremented by 1.

(c) If incrementing v's counter transformed it from 0 to 1 then v's list is scanned and each tuple on this list has its mark-counter decremented. If now some tuple s on v's list has its mark-counter equal to zero (i.e. it became good) then stage (b) above must be (recursively) applied to s.

Delete(t, R_i)

A. Delete t from the tree T_i in R_i.

B. For $1 \leqq m \leqq c$, if t was the only tuple on w_m's list and w_m's good-counter is zero, then delete w_m from C_{im}.

C. If t was good then the good-counter associated with v in R_{pi} is decreased (if it becomes zero and v's list is empty then v is removed from C_{pi}). If v's good-counter becomes zero then v's list is scanned and each tuple has its mark-counter incremented. If some tuple's mark-counter changes from 0 to 1 then the tuple is now bad and stage C of *Delete* has to be (recursively) applied to this tuple and R_p.

Addition analysis. Consider adding a tuple t into relation R_i where the database contains n tuples (we use the same notation as above). Entering t into T_i costs $O(\log n)$. Entering t into w_m's list (recall that $w_m = t[\mathbf{Z}_{im}]$ belongs to C_{im}) costs $O(\log n)$; as there are c such trees, the overall cost is $O(c \log n)$. The analysis of t's interaction (in case t is good) with R_p is a bit more intricate. First, the good-counter of v in C_{pi} has to be incremented at a cost of $O(\log n)$. Now, if as a result of this the counter has changed from 0 to 1, mark-counters for tuples on v's list are updated. This updating may cause some bad tuples in R_p to become good and the effect propagates up the tree.

The crucial point in the analysis is that the effect propagates on the unique path from R_i to the root and that in each relation node R along the way each tuple can lose at most one mark—the one corresponding to the unique child S of R which also lies on the path from R_i to the root. Suppose $t \in R$ becomes good. Using t's up-pointer, the list in the appropriate C-tree in R_i's parent can be accessed in $O(1)$ time. This list is then traversed and marks of tuples in the list are updated. Hence, since there are n tuples in the database, the overall cost of the propagation effect is $O(n)$. Summarizing, the overall cost of inserting t is $O(c \log n + n)$.

Deletion analysis. Finding t and deleting it from T_i and the lists on the C_{im} trees can be done in $O(c + \log n)$ time. However, if t was the only tuple on a list in C_{im} and the value w_m has a zero good-counter then w_m needs to be deleted ($O(\log n)$time). Thus the overall cost of updating T and the c trees is $O(c \log n)$. If t was bad we are done. Otherwise, v's good-counter in C_{pi} is decremented; if it becomes zero then, effectively, an R_i-mark is added to the tuples on v's list. If this transforms some tuples in R_p from good to bad, the effect might propagate up the tree. Again, the number of marks that can be added to all tuples in the database in the course of a single deletion is bounded by n. Hence, the overall cost of a single deletion is $O(c \log n + n)$.

THEOREM 5.1. *Let R be a relation in a tree database with n tuples, and let R have c children. Then a single tuple can be added or deleted from R in $O(n + c \log n)$ time.*

By the above theorem, any sequence of m operations during which the database never contained more than n tuples costs $O(mn)$. Another complexity measure is *amortized cost,* the cost of adding n tuples into an initially empty database. The main observation here is that in the course of n additions at most n tuples can become good and each tuple can lose at most all its marks. Thus the amortized cost for n additions (and no deletions) into a node with c children is $O(cn \log n)$. We summarize this by

THEOREM 5.2. *Consider a sequence of n additions to an initially empty database or n deletions and no additions applied to a database with n tuples. This sequence can be performed in* $O(\kappa n \log n)$ *time, where* κ *is the maximum number of children of a node in the qual tree.*

5.2. Additions into a general database. If the view attributes are not contained in any relation schema, or if the database is not a tree database, we transform the database and view to the previous case by adding new relations called *templates.* The problem of finding suitable templates will not be addressed here; see [18], [19]. One can think of templates as including in principle "all possible tuples". One way to achieve this is to let a template be total w.r.t. the database. This is fairly wasteful and we shall see other ways of maintaining templates in which only relevant tuples are maintained. In general, templates contain tuples which are computed in various ways from database relations; i.e., *template tuples* are generated from *original database tuples.*

PROPOSITION 1. *Let* $\mathbf{D} = (\mathbf{R}_1, \cdots, \mathbf{R}_k)$ *be a database and let S be a relation such that* $S \supseteq J(D)[\mathbf{S}]$. *Then for all views* $\mathbf{X}$,

$$\left(\mathop{\bowtie}_{i=1}^{k} R_i\right)[\mathbf{X}] = \left(\left(\mathop{\bowtie}_{i=1}^{k} R_i\right) \bowtie S\right)[\mathbf{X}]$$

(i.e. a view cannot be affected by adding S).

Proof. Apply elementary properties of the join operator. □

PROPOSITION 2. *The following example illustrates that populating a template S with less than* $(\bowtie_{i=1}^{k} R_i)[\mathbf{S}]$ *might produce incorrect views:*

Let $\mathbf{D} = (\mathbf{R}_1, \mathbf{R}_2)$, $\mathbf{X} = \mathbf{S} = \{\mathrm{B}\}$.

R_1:

A	B
1	2
3	4

R_2:

B	C
2	5
4	6

S:

B
2

Clearly, $(R_1 \bowtie R_2)[\mathbf{B}] \backslash S = \{\langle 4\rangle\}$; *also,* $(R_1 \bowtie R_2)[\mathbf{X}] = \{\langle 2\rangle, \langle 4\rangle\}$; *and* $((R_1 \bowtie R_2) \bowtie S)[\mathbf{X}] = \{\langle 2\rangle\}$; *therefore,* $((R_1 \bowtie R_2) \bowtie S)[\mathbf{X}]$ *is strictly contained in* $(R_1 \bowtie R_2)[\mathbf{X}]$. □

Consider first the case of a cyclic database in which the view attributes are contained in some relation; the other cases are similar. Assume the database was transformed into a tree database by adding some templates. For the good-bad mechanism to function, by Proposition 1 it is sufficient for each template S to contain $(\bowtie_{i=1}^{k} R_i)[\mathbf{S}]$ and by Proposition 2 it might not be sufficient for a template to contain less than that.

Next, we discuss various schemes for extending the good-bad mechanism to templates. Unlike relations where the "base set" of tuples is fixed, templates may undergo changes when base relation tuples are changed: the template base set may grow as a result of adding a tuple to the good set of a base relation, or shrink when such a tuple is deleted. The problem is parametrized according to the transformed schema structure, according to the following parameters:

τ: the number of templates;

γ: the maximum number of generators (defined below) per template;

κ: the maximum number of children of a node in the resulting qual tree.

Let $\mathbf{D}$ be a database schema transformed into a tree schema by adding τ templates. Consider the process of adding n tuples to an initially empty database state D. We separate the cost into two parts: that of finding the tuples to be entered into the templates, and that of entering all the tuples into the database; the latter consists of the cost of the addition of the n original tuples and the cost of adding template tuples, both using the good-bad mechanism. We have the following theorem.

THEOREM 5.3. *Adding n tuples into an initially empty transformed database with τ templates requires adding at most $O(\tau \cdot 2^n)$ tuples into templates.*

Proof. An addition of a tuple t into a relation R may introduce a "new value" $t[\mathbf{S}]$ for template S. Let $s = t[\mathbf{R} \cap \mathbf{S}]$. To enlarge the template,[5] we simply *duplicate S* and in one of the copies replace the $\mathbf{R} \cap \mathbf{S}$ columns with s. Thus, the addition of a tuple may double the number of tuples in each template.[6] The result follows since there are n original tuples and τ templates. □

COROLLARY. *Adding n tuples to an initially empty database requires at most $O(\kappa \tau n 2^n)$ time.*

Proof. By Theorem 5.3 $N = \tau 2^n$ tuples are added, and by Theorem 5.2 this costs $O(\kappa N \log N)$ time. □

The above result is discouraging since the cost is extremely high even for a small number of tuples. As we shall see, we can substantially improve this result.[7]

The manner in which templates are enlarged determines the cost of extending the good-bad mechanism. Let S be a template over attributes $\mathbf{S}$. One way to generate relevant S tuples is to join enough database relations to obtain all of S's attributes. Formally, the relations $\mathbf{R}_1, \cdots, \mathbf{R}_g$ are a *generator set* for S provided $\mathbf{S} \subseteq \bigcup_{i=1}^{g} \mathbf{R}_i$; they generate $S' = (\bowtie_{i=1}^{g} R_i)[\mathbf{S}]$. S' can then be partitioned to good(S') and bad(S') by the usual procedure. In other words, we have described a method for instantiating a candidate for containing both the good and the relevant bad tuples in a template. (See Fig. 5.2(a).)

The cost of tuple additions is dominated by the correct maintenance of templates, i.e. when a tuple is added to the good part of a generator relation, the templates for which it is a generator might have to be enlarged. This means joining the new tuple with all the other generators, a potentially costly procedure ($O(\tau n^{\gamma})$). Since there are at most n such additions, the overall cost is $O(\tau n^{\gamma+1})$. (A closer analysis reveals that the cost is $O(\tau/(\gamma-1^{\gamma-1})n^{\gamma})$.)

The following refinement reduces this cost. For each template we build a *generator tree* that is a full binary tree; the template is at its root and the generators at its leaves. An internal node consists of the join of its two child relations. (Note that the generator tree is a separate structure which comes in addition to the usual qual tree and the various balanced trees. See Fig. 5.2(b).)

In order to compute the cost of n additions into the generator relations of a template S, we make the following observations:

(1) When a tuple enters a generator relation, it has to be compared to its sibling in the generator tree in order to populate their parent.

(2) Each leaf v contains at most $\beta(v) = n$ tuples.

(3) The parent v of nodes v_1 and v_2 has at most $\beta(v) = \beta(v_1) \cdot \beta(v_2)$ tuples. Consequently, a node at distance h from the leaves has at most n^{2^h} tuples.

---

[5] Initially the template relation contains an arbitrary tuple.

[6] If duplicate tuples are eliminated from the template then its size cannot exceed $n^{|S|}$ and the term 2^n may be replaced by $n^{|S|}$.

[7] For $N = \tau \cdot n^{|S|}$, $O(\kappa \cdot |S| \cdot \log n \cdot n^{|S|})$ time is required and we will improve on that as well.

(4) The cost of adding $\beta(v_1)$ tuples to a child and $\beta(v_2)$ tuples to its sibling is exactly $\beta(v_1)\cdot\beta(v_2)$, the maximum size of their parent.

(5) The cost of all the additions into a set of generators is equal to the sum of the sizes of all the internal nodes of the generator tree.

THEOREM 5.4. *Suppose n tuples are added to an initially empty database. The time required to add all template tuples is* $O(\tau(n/\gamma)^\gamma)$.

Proof. First, consider two sibling nodes in the generator tree with a total of m tuples. The number of tuples in their parent node is maximum when each of the siblings has $m/2$ tuples. Therefore, the number of tuples in a generator tree is maximum when all its leaves have the same number of tuples. The worst case occurs when there are γ leaves and exactly n/γ tuples per leaf, in which case the total number of tuples is

$$\sum_{i=1}^{\log\gamma}\left(\frac{n}{\gamma}\right)^{2^i}\frac{\gamma}{2^i}=O\left(\left(\frac{n}{\gamma}\right)^{\gamma}\right).$$

Since there are τ templates, the total cost is $O(\tau(n/\gamma)^\gamma)$. □

COROLLARY. *Adding or deleting a single tuple to a database containing n original tuples requires at most* $O(\tau(n/\gamma)^\gamma+\kappa\gamma\log n)$ *time.*

COROLLARY. *Adding n tuples to an initially empty treefied database requires at most* $O(\kappa\tau/(\gamma^{\gamma-1})n^\gamma\log n)$ *time.*

This is more encouraging than the corollary to Theorem 5.3 since in many practical applications γ is small.

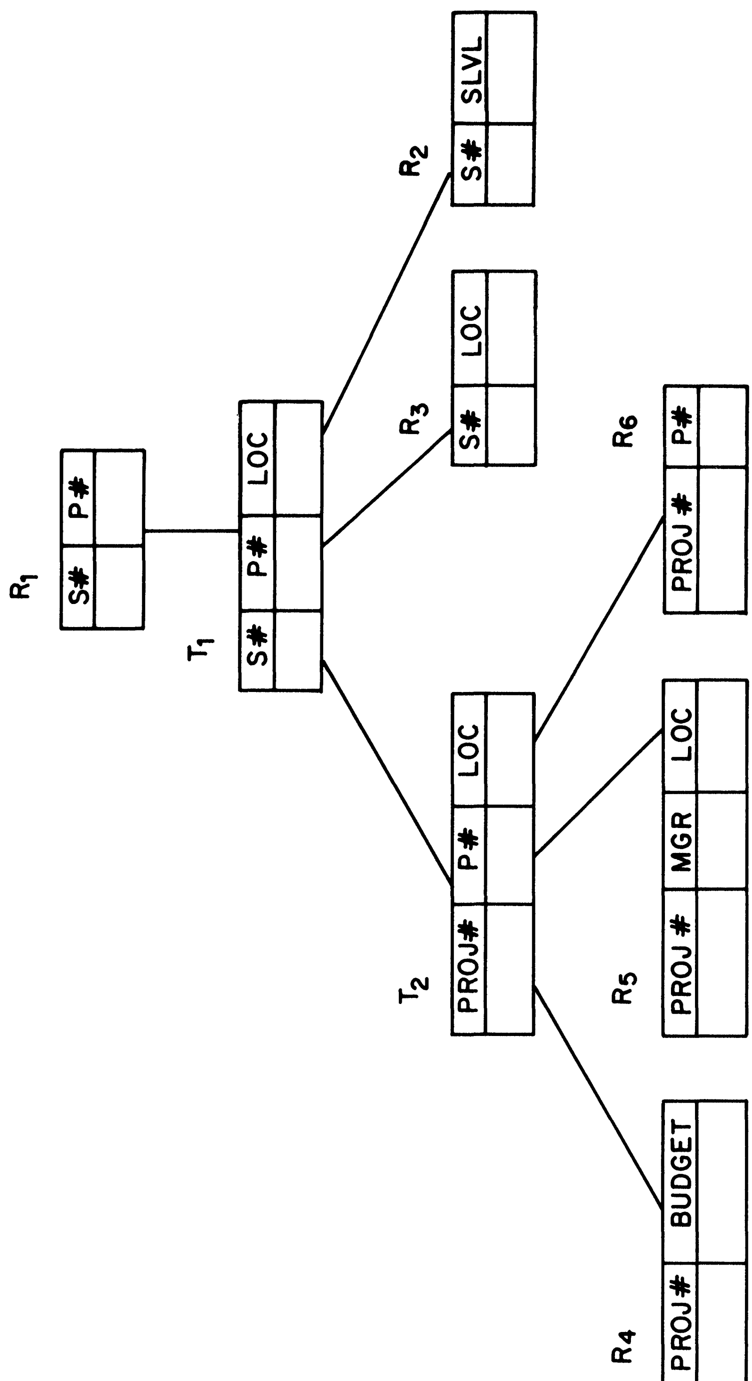
Finally, we note that the cost of a single deletion can be quite high, since it may cause many tuples in templates to become bad, costing the same as n additions. Practically, it seems better to do the following: each time we delete a tuple we also delete all tuples it helped generating (in templates). Thus, at all time, when n original tuples are in the database, there remain at most $O(\tau(n/\gamma)^\gamma)$ tuples in the database.

6. Conclusions. Several problems involving views were considered. It turns out that many view related problems are hard (Σ_2^p-complete) for arbitrary databases. Even when the database structure is relatively simple (tree databases), many problems remain NP-complete.

Each problem was treated for general databases and for the much simpler tree databases. We noticed the following "complexity reduction phenomenon"—NP-complete (Σ_2^p-complete) problems over general schemas become polynomial (NP-complete) over tree schemas. It is also interesting to note that while query processing over tree databases is polynomial, in the sense that intermediate results can be bounded by a polynomial in the input and the final result, such is not the case for view related problems. There seems to be an inherent "information loss" which makes view problems hard even on tree databases.

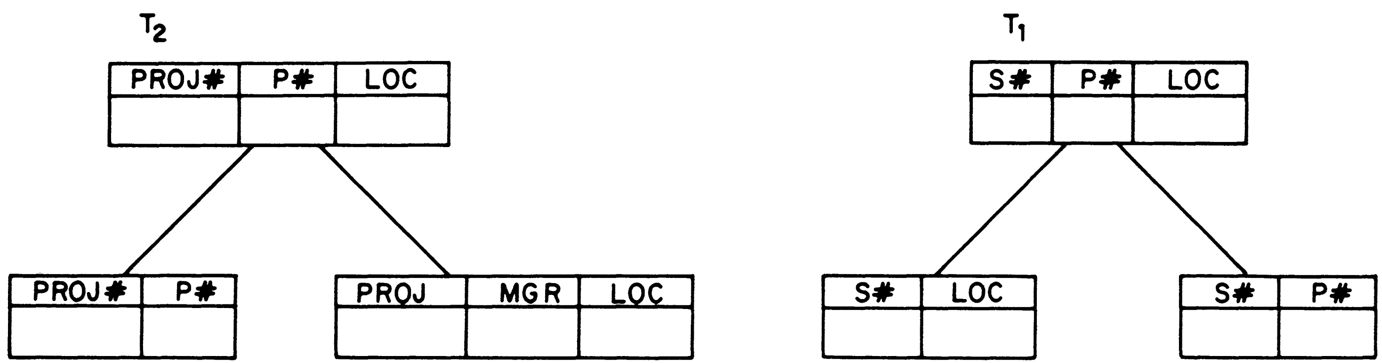
We have also examined view related problems over fixed schemas, in which only the data is allowed to vary. We have presented methods to handle this case. Their complexity is polynomial: for tree schemas the degree of the polynomial is independent of the schema structure, while for cyclic schemas the degree depends on the schema structure. Our results concerning fixed schemas are summarized in Table 6.1.

The log n factor arises from using balanced trees. We can eliminate it by using hashing, but then the results bound the average behavior, not the worst case. We do not know whether the bounds we found are tight and we leave it as an open problem. This paper also suggests additional problems such as maintaining multiple views, and that of extending the mechanism to an off-line sequence of updates to base relations.



(a) Adding templates T_1 and T_2 to the original schema.

(b) Generator trees.



R_1, R_2, R_4, R_5: are as in Fig. 5.1;
R_3: supplier S# is at location LOC;
R_6: project PROJ# needs part $P\#$;
T_1, T_2: added templates.
The view is on $S\#$ and $P\#$.

FIG. 5.2. *Adding templates.*

TABLE 6.1

	A single addition or deletion	A sequence of n additions
Tree databases	$O(\kappa n \log n)$	$O(n \log n)$
Cyclic databases	$O\left(\kappa \frac{\tau}{\gamma^{\gamma-1}} n^{\gamma} \log n\right)$	$O\left(\frac{\tau}{\gamma^{\gamma-1}} n^{\gamma} \log n\right)$

The complexity measure used in the analysis was the number of tuple operations. Thus the analysis is directly applicable to small scale databases whose data, or very large portions thereof, fits into memory. The tuple operations measure is inadequate for large databases in which only a small portion of the data can reside in main memory. Consider a large scale database environment. First, the balanced trees may be implemented as B-trees or replaced by a suitable hashing scheme. Second, recursive add and delete operations should be made to recurse on sets of tuples rather than on single tuples. This reduces the number of relations that are accessed at any one time, a better use of buffers is achieved and therefore secondary storage access performance is improved.

Acknowledgments. We thank R. Pinter for commenting on an earlier version of the manuscript and S. Even for simplifying the proof of Lemma 4.1.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1976.

[2] C. BERGE, *Graphs and Hypergraphs*, North-Holland, Amsterdam, 1973.

[3] P. A. BERNSTEIN AND D. W. CHIU, *Using semi-joins to solve relational queries*, J. Assoc. Comput. Mach., 28 (1981), pp. 25-40.

[4] C. BEERI, R. FAGIN, D. MAIER, A. MENDELZON, J. D. ULLMAN AND M. YANNAKAKIS, *Properties of acyclic database schemes*, in Proc. 13th Ann. ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, May 1981, pp. 355-362.

[5] C. BEERI, R. FAGIN, D. MAIER AND M. YANNAKAKIS, *On the desirability of acyclic database schemes*, J. Assoc. Comput. Mach., 30 (1983), pp. 479-513.

[6] P. A. BERNSTEIN AND N. GOODMAN, *The power of natural semijoins*, this Journal, 10 (1981), pp. 751-771.

[7] A. K. CHANDRA AND P. M. MERLIN, *Optimal implementation of conjunctive queries in relational databases*, in Proc. 9th Ann. ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, May 1977, pp. 77-90.

[8] E. F. CODD, *A relational model for large shared data banks*, Comm. ACM, 13 (1970), pp. 377-387.

[9] S. S. COSMADAKIS, *The complexity of evaluating relational queries*, in Proc. ACM SIGACT-SIGMOD Conference on Principles on Database Systems, Atlanta, March 1983, pp. 149-155.

[10] U. DAYAL AND P. A. BERNSTEIN, *On the updatability of relational views*, in Proc. 4th VLDB Conference, West Berlin, September 1978.

[11] R. FAGIN, *Degrees of acyclicity for hypergraphs and relational database systems*, J. Assoc. Comput. Mach., 30 (1983), pp. 514-550.

[12] R. FAGIN, A. O. MENDELZON AND J. D. ULLMAN, *A simplified universal relation assumption and its properties*, ACM Trans. Database Systems, 7 (1982), pp. 343-360.

[13] M. H. GRAHAM, *On the universal relation*, Technical Report, Univ. Toronto, September 1979.

[14] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of* NP-*completeness*, W. H. Freeman, San Francisco, 1979.

[15] N. GOODMAN AND O. SHMUELI, *Tree queries: a simple class of queries*, ACM Trans. Database Systems, 4 (1982), pp. 653-677.

[16] ———, *Syntactic characterizations of tree database schemas*, J. Assoc. Comput. Mach., 30 (1983), pp. 767-786.

[17] ———, *The tree projection theorem and relational query processing*, J. Comput. System Sci., 28 (1984), pp. 60–79.

[18] ———, *Transforming cyclic schemas into trees*, in Proc. ACM SIGACT-SIGMOD Conference on Principles of Database Systems, Los Angeles, CA, March 1982, pp. 49–54.

[19] N. GOODMAN, O. SHMUELI AND Y. C. TAY, GYO *reductions, canonical connections, tree and cyclic schemas and tree projections*, J. Comput. System Sci., 29 (1983), pp. 331–349.

[20] R. HULL, *Acyclic join dependencies and database projections*, J. Comput. System Sci., 27 (1983), pp. 231–349.

[21] P. HONEYMAN, R. E. LADNER AND M. YANNAKAKIS, *Testing the universal instance assumption*, Inform. Process. Lett., 10 (1980), pp. 14–19.

[22] D. MAIER AND J. D. ULLMAN, *Connections in acyclic hypergraphs*, in Proc. ACM SIGACT-SIGMOD Conference on Principles of Database Systems, Los Angeles, CA, March 1982, pp. 34–39.

[23] ———, *Maximal objects and the semantics of universal relation databases*, ACM Trans. Database Systems, 8 (1983), pp. 1–14.

[24] N. SPYRATOS, *Translation structures of relational views*, in Proc. 6th VLDB Conference, Montreal, 1980.

[25] O. SHMUELI AND A. ITAI, *Maintenance of views*, in Proc. SIGMOD 1984, Boston, MA, June 1984, pp. 240–255, SIGMOD Record Vol. 14, No. 2.

[26] R. E. TARJAN AND M. YANNAKAKIS, *Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs*, this Journal, 13 (1984), pp. 566–579.

[27] J. D. ULLMAN, *Principles of Database Systems*, Computer Science Press, Potomac, MD, 2nd ed., 1983.

[28] M. YANNAKAKIS, *Algorithms for acyclic database schemes*, in Proc. 7th VLDB Conference, 82–94, Cannes, September 1981.

[29] C. T. YU AND M. Z. OZSOYOGLU, *An algorithm for tree-query membership of a distributed query*, in Proc. COMPSAC79, IEEE Comp. Society, November 1979.